Variables, Strings, and Loops

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonce

Department of Computer Science and Engineering

University at Buffalo

Administrivia

Have you done your assigned reading?

Can you connect to Emon?

Did you compile and run Hello World?

Remember that many of you are new to the command prompt!
Check the Piazza post on this.

Read everything for all assignments and labs.

Check Piazza frequently.

Things you already know: expressions

An expression is a part of a program that has a value.

Expressions can be simple or compound

simple	compound
`4`	`3 + 4`
`4.5`	`4.5 * 3.7`
`x`	`x++`
`'a'`	`'a' + 1`
`"a"`	`strlen("a")`
`true`	`y = 3`

Expressions may have side effects (*e.g.* `x++` and `y = 3`).

Things you already know: statements

A statement is a part of a program that:

- has a side effect
- does not have a value

Statements can be simple or compound.

simple	**compound**
`y = 3;`	`if (x<y){...} else {...}`
`puts("a");`	`while (x<y){...}`
`x++;`	`for (int x=0; x<len; x++){...}`

Things you already know: values

Every expression has a value.

Values can be simple or compound.

simple	**compound**
3	*an array*
`'c'`	*a struct*

Things you already know: types

Each value and variable is associated with a type.

The type determines:

- **size:** the number of bytes occupied by a value ¶
- **representation:** how a value is encoded as bits
- **operations:** which operators are valid with a value

minimal

```
// A minimial C program
// Execution of your code starts in the 'main'
   function
int main() {
    return 0;
}
```

puts

```
#include <stdio.h>

// Hello World
int main() {
    puts("Hello World"); // string literals are
       delimited by double quotes
    return 0;
}
```

delimiters

```
#include <stdio.h>

// Hello World
int main() {
    puts('Hello World'); // single quotes delimit
       single characters
    return 0;
}
```

putchar

```
#include <stdio.h>

int main() {
    putchar('H');
    putchar('i');
    putchar('\n');
    putchar(72);
    putchar(105);
    putchar(10);
    return 0;
}
```

printf

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

printf specifiers

```
#include <stdio.h>
int main() {
    int i = 42;
    printf("The value of i: %d\n",i);
    printf("The value of i: %x\n",i);
    double d = 2.71828;
    printf("The value of d is %f\n",d);
    printf("The value of d is %d\n",d);
    char letter = 'X';
    printf("Hey, '%c' marks the spot.\n", letter);
    char name[] = "River";
    printf("%s said, \"No power in the 'verse can
       stop me\"\n",name);
    return 0;
}
```

command-line argument

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    printf("Hello, %s!\n", argv[1]);
    return 0;
}
```

command-line arguments

```
#include <stdio.h>

int main(int argc, char * argv[]) {
    for (int i=0; i<argc; i++) {
        printf("argv[%d] is \"%s\"\n", i, argv[i]);
    }
    return 0;
}
```

cse-strlen1

```
#include <stdio.h>

int cse_strlen(char str[]) {
    int len = 0;
    for (int i=0; str[i] != '\0'; i++) {
        len = len + 1;
    }
    return len;
}
```

cse-strlen2

```
#include <stdio.h>

int cse_strlen(char str[]) {
    for (int i=0; str[i] != '\0'; i++)
        ;
    return i;
}
```

cse-strlen3

```
#include <stdio.h>

int cse_strlen(char str[]) {
    int i;
    for (i=0 ; str[i] != '\0'; i++)
        ;
    return i;
}
```

Types

C is a typed language.

Every variable has a type, and is declared.

Every value assigned to a variable must match that type.

The compiler will automatically convert between some types.[1]

Valid:

```
int x = 5;
float y = 2.0;
x = 37.0;
y = x;
```

Invalid:

```
int x = 0;
x = "Hello, world!";
```

[1] Dennis M. Ritchie (DMR) said “C is strongly typed, but weakly enforced.”

Some Types

There are many types; for now, consider:

- `int`: Integers of a convenient size for the computer (32-bit for us)
- `char`: Characters (typically 8-bit integers)
- `double`: Double-precision floating-point numbers

There are also array types.

Array types are declared with square brackets: `[]`:

- `char a[]`: An array of `char` variables. Often used for C strings.
- `int scores[200]`: An array of exactly 200 `int` variables.

Declaring Variables

Variables are declared by stating their type and name.

```
int x;              /* x is an integer */
double d;           /* d is a floating-point double */
```

Various modifiers can be applied to variables.

In particular, `const` declares the variable to be a constant. A `const` variable can only be assigned a value in its declaration.

Scope, part 1

Variables in C have scope: the part of a program where the variable can be used.

Scope is determined by how and where the variable is declared.

The following are possible scopes in C:
global, file-local, and local.

A variable cannot be used out of scope.

Scope, part 2

Variables declared outside of any block ({}):

- are normally global: they can accessed by any code ¶must be declared `extern` in other files
- are file-local with the modifier `static`: they can be accessed by any code in this file

Variables declared in a block:

- Come into scope where declared¶
- are valid until the scope's } or end-of-file

Lifetime

Variables in C have lifetime: the period of time during the execution of a program that the variable exists in memory.

For many variables, their lifetime is as long as their scope exists.

So far:

- Global variables have a lifetime of “forever”
- Local variables have a lifetime of “while in scope”

Arrays

C arrays are a series of contiguous memory locations.
(This will become important later.)

Arrays are declared with `[]`. The size is between `[]`.

Every array has a fixed size, however array declarations can have three "sizes", depending on what's in the `[]`:

- Unknown size: Nothing is specified
- Constant size: A constant expression is specified
- Variable size: A run-time computed expression is specified

Array Sizes

Array sizes specify how many elements are in the array.

```c
int x[32];
int matrix[32][16];
```

C does not remember the array's size.¶

This means that illegal accesses aren't caught.[2]

```c
int x[4];
x[10234] = 0;           /* Whoops. */
```

[2] If you're lucky, you might get a warning about uninitialized access.

Static Initializers

An array can be initialized all at once at declaration.

```
int array[10] = { 0, 3, 5, 0, 0,
                  1, 0, 0, 2, 0 };
```

This is called a static initializer.

Static initializers can be used only at declaration.

```
int array[3];

array = { 1, 3, 5 };    /* syntax error */
```

C Strings

C strings are just arrays.

Strings, as they are arrays, are not associated with a length. (You have to count the characters to know how long they are.)¶

A C string consists of:

- the characters in the string, followed by
- a zero byte (the ASCII NUL character) (NUL terminator).

The zero byte is idiomatically written `'\0'`.

ASCII

ASCII[3] is a mapping of numbers to characters.

C strings can be in many encodings, but C code is in ASCII.¶

ASCII contains Latin characters, numbers, and punctuation.

The Unix manual page at `man 7 ascii` describes the mapping.

[3] American Standard Code for Information Interchange

Quoted Strings

Quoted strings automatically build such arrays.

```
char str[] = "Hello";
char str[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

A quoted string may be assigned to an array only at declaration.

After declaration, quoted strings must be copied into arrays:

```
char str[32];

strncpy(str, 32, "Hello");  /* See man 3 strncpy */
```

String Functions

There are many string functions in the C library.

Most of them are defined in `<string.h>`.

Some useful examples:

- `strlen()`: Compute the length of a string by counting bytes
- `strncpy()`: Copy a string until its NUL character
- `strncat()`: Concatenate one string to another
- `strstr()`: Search for one string inside another

Strings as Pointers

The idiomatic string type is `char *`.

Arrays and pointers are closely related, we'll discuss this later.

```
char *str = "Hello, CSE 220";

char array[] = "Another string";
char *otherstr = array;
```

Character Constants

An ASCII character can be interpreted as an integer with `''`.

```
char c = 'A';              /* 65 */
int i = 'B';               /* 66 */
```

Each byte of a string can be assigned in this fashion.

```
char str[] = "emacs";
/* Give it the respect it deserves */
str[0] = 'E';
```

The for Loop

The C for loop is its most versatile loop.

It allows looping over almost anything.

```
for (initialization; condition; increment) {
    body;
}
```

It translates to a more traditional while loop (with caveats):

```
initialization;
while (condition) {
    body;
    increment;
}
```

Looping over Arrays

A common use of the `for` loop is looping over arrays:

```
int array[ARRAYSZ];

for (int i = 0; i < ARRAYSZ; i++) {
    /* Use array[i] */
}
```

Remember that you must somehow know the size of the array.

Looping over Strings

It is idiomatic to loop over strings:

```
for (int i = 0; str[i] != '\0'; i++) {
    /* use str[i] */
}
```

Note that the string length is never directly computed!

Strings, Arrays, and Loop Example

We will develop `strlen()` together.

Summary

- C is a typed language
- Every variable has a type
- Variable values must match the type
- Variables have scope, and cannot be used outside that scope
- Arrays are contiguous memory locations
- Array syntax uses [ ]
- C strings are arrays of characters
- Every C string is terminated with a zero byte
- For loop syntax
- For loops are very flexible

Next Time …

- Boolean values
- Conditional statements
- Control flow

References I

Required Readings

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 1: 1.9, 1.10; Chapter 2: Intro, 2.1–2.4. Prentice Hall, 1988.

License